Workshop/Summer School on Evolutionary Computing
Lecture Series by Pioneers, August 18-22, 2008, Londonderry

Evolutionary Software Improvement for Instruction Set Meta-evolution

Michael Orlov and Moshe Sipper
Department of Computer Science, Ben-Gurion University
PO Box 653, Beer-Sheva 84105, Israel
Email: {orlovm,sipper}@cs.bgu.ac.il

***Abstract*—A major drawback of evolutionary computation is that only relatively small-scale solutions can be found during the search process—it is currently impossible to evolve a whole chunk of software. We propose a method named *evolutionary software improvement*, which makes it possible to take some module, or some aspect of an existing software system, and improve just that part using evolutionary computation. We apply evolutionary software improvement to our genetic-programming system Megavac, refining its instruction set for a class of multi-input handling problems. This allows us to develop problem-fitting instruction sets in a meta-circular fashion, and leads to surprising solutions evolved by Megavac for the given class of problems.**

I. Introduction

A major drawback of evolutionary computation is that only relatively small-scale solutions can be found during the search process—it is currently impossible to evolve a whole chunk of software. We propose a method named *evolutionary software improvement*, which makes it possible to take some module, or some aspect of an existing software system, and improve just that part using evolutionary computation.

The choice of a system on which to test and refine the software improvement process outlined above is critical. The system must be big enough so that it is not trivial, and can be improved in ways that are both interesting and unexpected. The system must not be *too* big, otherwise its numerous architectural and implementation details will impede the testing and the tweaking of the process. The system's properties must be easy enough to change, to allow focusing on the important properties of evolutionary software improvement. This implies sufficient researcher familiarity with the system.

Our own evolutionary-computation platform *Megavac*, described in Section III, is a good candidate for such a software system. It is a spatially structured steady-state evolutionary system, with individuals represented as cyclic linear programs. It has interesting properties, such as ability of individuals to send and receive messages. The system affords easy definition of properties that can be improved, such as time of convergence to best individual, or quality of best individual. It is non-trivial—improvements to, e.g., the main loop, can result in interesting variants of evolutionary algorithms. Finally, Megavac has 18 000 lines of code, so the system itself is non-trivial.

M. Orlov is partially supported by the Lynn and William Frankel Center for Computer Sciences.

The self-referential nature of using evolutionary software improvement on an evolutionary-computation platform is doubly beneficial, since interesting results not only have the potential to make Megavac human-competitive, but also provide novel results in the evolutionary-computation field.

The instruction set employed in genetic programming has a huge impact on the evolutionary process. Ofria et al. [1] have shown that even no-op instructions have vital effect. The presence of some instructions may have detrimental effect on the search space. On the other hand, the presence of complex meta-instructions that are equivalent to a sequence of simpler instructions may help solve the specific class of problems at hand. In this paper we develop problem-fitting instruction sets in meta-circular fashion.

In Section IV, we apply meta-evolution to a combined ECJ-Megavac framework. A subset of the complete Megavac instruction set is evolved in order to facilitate faster evolutionary solutions of multi-input problems using concurrent layered learning. Such problems require handling multiple inputs, leading to surprising solutions. This process offers a glimpse into the area of evolutionary software improvement, outlined in Section V.

II. Evolutionary Software Improvement

A. Area of Application

In order to make evolutionary software improvement a feasible process for an existing software system, the methodology must satisfy several requirements.

First of all, there must be some aspect of the system that can be changed in order to improve some of the system's characteristics. This aspect is not necessarily a specific component; it can be some behavioral aspect of the system as a whole—for instance, the main loop.

Second, since we would like to use evolutionary computation to improve the chosen sub-system, its function needs to be representable as an evolvable program. This necessitates definition of suitable primitives that are sufficiently expressive and allow the necessary functionality to evolve.

Lastly, the chosen component or aspect has to be amenable to evolutionary improvement. On the one hand, the functionality should be sufficiently algorithmic in nature to allow better behavioral process to evolve. On the other hand, a comparison of evolved functionalities should be reasonably fast to allow improved behavior to emerge in the relevant time frame. The

latter constraint does not, however, restrict the size of the system as a whole—it is a fair assumption that a big software system has the capacity to test varying functionalities of its components without long startup and shutdown sequences.

B. The Software Improvement Process

The system requirements outlined above allow us to define the following generic process for evolutionary software improvement.

Initially, we need to analyze the software system at hand and locate a behavioral aspect, a component, a module, or a package that can be expressed algorithmically. The selected component has to be sufficiently independent to allow expression with a program of reasonable size. It must possess sufficient behavioral freedom to justify the evolutionary approach. Substitution and evaluation of an altered component must be sufficiently brief in order to allow better components to develop.

We then need to define a fitness measure that quantifies the performance of the component. The fitness must express objective software improvement goals. Objectives such as efficiency, quality, and parsimony pressure, need to be considered.

Finally, we should analyze the chosen component, and define the language for expressing evolving individuals. Primitives must allow the necessary freedom of expressed individuals, and the existing component must be naturally expressible in the language. The initial population is then initialized with programs expressing the existing component.

If the software improvement process evolves a better aspect or component, according to the chosen fitness measure, then the system as a whole is improved using evolutionary computation. Moreover, if such a system is a state of the art software, the result may be human-competitive [2].

III. Evolutionary Platform Megavac

Megavac is a spatially structured, steady-state evolutionary platform, designed by Orlov and Sipper, with individuals represented as cyclic linear programs. It is similar in concept to Avida [3], but the emphasis is on evolutionary computation and not on artificial life. Thus, for instance, the programs do not need to replicate themselves, and fitness is expressed directly, and not via metabolism speed. Megavac is modular, and its many aspects are easily configurable. It was designed to explore emergent cooperation between spatially structured individual programs.

The main components of Megavac are: genomes container, instruction scheduler, connection topology, selection method, mutator, reproducer, and the environment. The genomes are assembly programs with variable-length code, registers, data and control stacks, connectors, and mutation probabilities. Each genome can be in two states: *wait* and *active*. A genome that executes a `wait` instruction goes into *wait* state, and remains in this state until it receives a message from the environment, or from a neighbor (which uses a `send` instruction to send messages). Once a genome answers a message, it can be rewarded with fitness, either implicitly by the environment, or explicitly by the neighbor. Thus, a capability developed against the environment can be potentially exploited by its neighbors. Genomes with low fitness are replaced using the configured selection and reproduction strategies.

Linear genetic programming [4] is used for genome representation. The main advantage of this representation is that genomes can be evaluated step-by-step in parallel, allowing for more freedom in features such as inter-genome communication. Another major benefit of linear genetic programming in the context of evolutionary software improvement is the high execution speed achievable with this straightforward implementation. The high execution speed makes it possible to evaluate a population of Megavac configurations in reasonable time. A more complete description is not possible here due to space restrictions.

IV. Experiments

A. Experimental Setup

The experimental setup for our evolutionary software improvement consisted of evolving bit vectors using the ECJ [5] framework, each vector representing a subset of the complete Megavac instruction set. The complete instruction set is shown in Table I.

The ECJ part of the meta-evolution setup consisted of a genetic algorithm with bit-vector individuals, each bit enabling or disabling a Megavac instruction. A population size of 40 was used, and the genetic algorithm was run for 40 generations. For all problems we applied a single-point crossover probability of 0.8, a single bit mutation probability of 0.05, and tournament selection of size 2.

Each meta-evaluation consisted of running Megavac for 1000 generations, each generation consisting of 32 instruction execution rounds. A population size of 128, torus four-neighbor topology, and tournament selection that included all neighbors, were used for all problems. Variable-length mutation with initial code segment size of 1, data and control stack sizes of 4, and 3 general-purpose registers were also specified.

The fitness of a single Megavac execution was defined as the area below the maximal fitness curve—that is, the sum of per-generation maximal fitnesses. This definition has the nice property of being independent from problems on which Megavac is run. Other possibilities include, e.g., area below the average fitness curve, and sum of fitness exponents, in order to emphasize higher Megavac fitnesses.

Since Megavac employs linear genetic programming with simple instructions, meta-evolution proceeds reasonably fast. A single run with the above settings completes within 22 minutes on a 2.6 GHz dual-core AMD Opteron workstation, when ECJ uses two threads for evaluation. Since ECJ easily supports parallelization of the evolutionary process in a network, this integrated framework should easily scale to more resource-demanding experiments.

TABLE I
THE COMPLETE MEGAVAC INSTRUCTION SET. ALL VALUES ARE FLOATING POINT, AND THERE ARE TWO STACKS: DATA AND CONTROL. IN ADDITION TO THE ACCUMULATOR REGISTER, THERE ARE A NUMBER OF GENERAL-PURPOSE REGISTERS. COMMUNICATION INSTRUCTIONS ASSUME THAT THE DEFAULT CONNECTOR INDEX RESIDES IN THE FIRST GENERAL-PURPOSE REGISTER.

Instruction	Description
	ALU instructions
`zero`	Zero the accumulator register
`add`	Pop value from data stack, and add to accumulator
`sub`	Pop value from data stack, and subtract from accumulator
`neg`	Negate the value in accumulator
`mul`	Multiply accumulator by popped data stack value
`div`	Divide accumulator by popped data stack value
`rnd`	Put a Gaussian random value $\mathcal{N}(0,1)$ in accumulator
`erc` *value*	Put ephemeral random constant *value* drawn from $\mathcal{N}(0,1)$ in accumulator
	Memory instructions
`push`	Push accumulator value into data stack
`pop`	Pop value from data stack into accumulator
`drop`	Pop value from data stack
`dup`	Push top data stack value into data stack
`swap`	Exchange accumulator with data stack top
`c2d`	Pop value from control stack, and push it into data stack
`d2c`	Pop value from data stack, and push it into control stack
	Register instructions
`load` *index*	Copy general-purpose register *index* to accumulator
`store` *index*	Copy accumulator to general-purpose register *index*
`rswap` *index*	Exchange accumulator with general-purpose register *index*
	Control instructions
`jump` *offset*	Jump *offset* instructions from the current code position
`call` *offset*	Push next instruction position into control stack, and jump *offset* instructions
`ret`	Pop code position from control stack, and jump to that position
`brnz` *offset*	If accumulator is non-zero, jump *offset* instructions
`brgez` *offset*	If accumulator is non-negative, jump *offset* instructions
`brlez` *offset*	If accumulator is non-positive, jump *offset* instructions
`brge` *offset*	If accumulator is not less than data stack top, jump *offset* instructions
`brle` *offset*	If accumulator is not greater than data stack top, jump *offset* instructions
	Communication instructions
`send`	Send accumulator to default connector index
`sendn` *connector*	Send accumulator to *connector* index
`wait`	Wait for data message or environment input in mailbox, and for input connector index in default connector register
`waitn`	Similar to `wait`, but ignore environment input
`fitness`	Send and subtract fitness given by accumulator value to default connector
`read`	Read message from mailbox into accumulator
	Miscellaneous instructions
`nop`	Do nothing

B. A Multi-input Problem

Megavac facilitates concurrent layered learning [6] via *composite environments*. Here we define an environment combining `Echo`, which rewards genomes for returning the same input to the environment, and `SubTwo`, which rewards genomes for returning the difference of two previous inputs. `Echo` rewards genomes with a fitness of 3.0, and `SubTwo` rewards with 15.0. As a rule, more complex tasks should have exponentially higher reward, to compensate for the longer required genome length. This is important, since genomes are executed circularly, and rewards collected during a single generation are combined.

TABLE II
INSTRUCTION SETS EVOLVED DURING EVOLUTIONARY SOFTWARE IMPROVEMENT OF MEGAVAC. BEST-OF-RUN INSTRUCTION SETS ARE SHOWN. META-FITNESS IS THE AREA BELOW THE MAX-FITNESS CURVE IN EACH MEGAVAC RUN. AVERAGE MEGAVAC FITNESS IS DERIVED FROM DIVIDING THE META-FITNESS BY THE NUMBER OF MEGAVAC GENERATIONS, 1000. AVERAGE FITNESS OF OVER 40.0 GUARANTEES (A VERY GOOD) ABILITY TO SOLVE `SUBTWO`.

Meta-fitness	Average	Instruction set
79104	79.1	`brge`, `brlez`, `brnz`, `call`, `dup`, `fitness`, `nop`, `pop`, `push`, `read`, `rswap`, `send`, `sub`, `swap`, `wait`, `zero` *(16 instructions)*
74278	74.3	`c2d`, `dup`, `erc`, `fitness`, `pop`, `push`, `read`, `rnd`, `rswap`, `send`, `store`, `sub`, `swap`, `wait`, `zero` *(15 instructions)*
82820	82.8	`add`, `brge`, `brlez`, `drop`, `dup`, `erc`, `jump`, `pop`, `read`, `send`, `sendn`, `sub`, `swap`, `wait` *(14 instructions)*
82742	82.7	`add`, `brge`, `call`, `drop`, `jump`, `load`, `nop`, `push`, `read`, `rswap`, `send`, `sendn`, `store`, `sub`, `wait`, `waitn` *(16 instructions)*
79348	79.3	`brge`, `brgez`, `brnz`, `c2d`, `d2c`, `erc`, `fitness`, `neg`, `nop`, `push`, `read`, `ret`, `rswap`, `send`, `store`, `sub`, `wait`, `waitn` *(18 instructions)*

Without `Echo`, `SubTwo` is impossible to evolve, since minimal solution length is six, and there are no approximate solutions of shorter lengths. Moreover, reaching that solution in the composite environment when using the complete instruction set requires a huge number of generations. We did not succeed in evolving a solution to *SubTwo* in any run of 10 000 generations when enabling the complete instruction set. This is due to the large dimension of the search space of 33 instructions. For instance, in the first generation of the meta-evolution, a typical Megavac execution will only solve `Echo` in the middle stages of the 1000 generations.

We ran a total of five experiments, aiming to evolutionarily improve Megavac's performance on the `Echo+SubTwo` problem. Table II shows the results of these experiments. The first instruction set in Table II represents a typical run of the evolved framework. Early on, individuals containing `wait-read-send` as a subsequence, solving `Echo`, appear in the population. Immediately after such ideal individual appears in generation 52, non-optimal individuals for solving `SubTwo` begin to appear, containing multiple superfluous instructions. Starting with generation 257, the population contains an optimal individual `wait-read-push-rswap(2)-sub-send`. It is a surprising result, since the genome reads an input only once per each output, although it succeeds in producing a

difference between two inputs to the environment. Another optimal individual frequently contained in meta-evolutionary runs results is `wait-read-swap-dup-sub-send`.

The evolved instruction subset which we considered in detail is: `brge`, `brlez`, `brnz`, `call`, `dup`, `fitness`, `nop`, `pop`, `push`, `read`, `rswap`, `send`, `sub`, `swap`, `wait`, `zero`. This instruction set includes only 16 instructions of the total possible 33, a typical result in our evolutionary experiments. Thus, the evolutionary software improvement process can be seen to weed out unnecessary, or at least less contributing, instructions, and improving the software system as a whole by reducing its complexity and tightening its code.

C. Validating the Reduced Instruction Set

In order to test the applicability of evolutionary software improvement, described in Section IV-B, we added another multi-input problem to the composite environment. The additional problem is `SubSq`, awarding a fitness value of 75.0 to genomes successfully producing output $x^2 - y$ for inputs x and y.

With the complete instruction set of size 33, Megavac did not evolve a solution to `SubSq` in any run of under 10 000 generations. This is not surprising, since a solution to the simpler problem of `SubTwo` does not appear in a typical run with similar number of generations when all instructions are enabled. Since a genome solving `SubSq` is expected to evolve from a genome solving `SubTwo`, it is unreasonable to expect a solution to `SubSq` to develop reasonably fast when the complete instruction set is used.

However, when the instruction set was restricted to the subset analyzed in Section IV-B with the additional `mul` instruction enabled, running Megavac with the same settings after adding `SubSq` to the composite environment produced the ideal individual `wait-read-swap-dup-push-mul-sub-send` at generation 2066. We omit other experimental details due to space constraints.

V. Discussion and Future Work

We have shown the feasibility of *evolutionary software improvement* on our evolutionary system Megavac. Representing Megavac as a genetic program, and evolving it using traditional methods would not be possible. Instead, we located a critical component affecting Megavac's evolutionary performance—its instruction set, represented this component as an individual in a genetic algorithm, and evolved instruction subsets that drastically improved the performance of Megavac. This performance was then validated on an extended problem set. Why those specific instructions were chosen during the meta-evolutionary process of software improvement is interesting, and left for future research.

We view evolutionary software improvement primarily as a general technique for applying evolution to complex systems. In this work we have evolved Megavac's instruction set. The next step is applying the more capable mechanism of genetic programming to meta-evolution. We are currently working on other aspects of the framework that can be evolutionarily improved.

One such aspect is expressing the evolutionary process as an algorithm. The evolutionary computation field has in its toolbox numerous techniques for guiding the evolutionary process, beyond the literal select-vary-reproduce approach. Expressing Megavac's reproduction process as an algorithm, and evolving this algorithm using evolutionary computation can provide objective insight into existing techniques, and hopefully automatically develop new ones. For instance, memetic algorithms [7] achieve excellent results by combining evolutionary exploration with local search. Will an automatically developed process possess similar properties? Successful formulation of the evolutionary process as an evolvable algorithm requires careful definition of primitives, such as reproductive variation operators.

Another aspect is when and how inputs are provided to the genomes. When an input is provided to a genome by the environment, inputs more suitable to certain tasks might better advance the evolutionary process. The decision of which inputs to provide can depend on the tasks a genome has already solved. On the other hand, the environment can abstain from providing an input to some genome altogether, if that genome could be used by other genomes via a message system. The decision of whether to abstain can also be developed evolutionarily. Can evolutionary development of input generation behavior facilitate the emergence of inter-genome cooperation?

The evolution of the aspects above offers a glimpse into feasibility of the area of evolutionary software improvement.

References

[1] C. Ofria, C. Adami, and T. C. Collier, "Design of evolvable computer languages," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 4, pp. 420–424, Aug. 2002.

[2] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer, Jul. 2003.

[3] C. Adami and C. T. Brown, "Evolutionary learning in the 2D artificial life system 'Avida'," in *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, July 6–8, 1994, Cambridge, Massachusetts, USA*, R. Brooks and P. Maes, Eds. MIT Press, Sep. 1994, pp. 377–381.

[4] M. F. Brameier and W. Banzhaf, *Linear Genetic Programming*, ser. Genetic and Evolutionary Computation. Springer, Dec. 2006.

[5] S. Luke and L. Panait, "A Java-based evolutionary computation research system," Internet site, Mar. 2004. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[6] S. Whiteson and P. Stone, "Concurrent layered learning," in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, July 14–18, 2003, Melbourne, Australia*, J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, Eds. ACM Press, Jul. 2003, pp. 193–200.

[7] N. Krasnogor and J. Smith, "A tutorial for competent memetic algorithms: Model, taxonomy, and design issues," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 5, pp. 474–488, Oct. 2005.